

Redes Neuronales y Aprendizaje profundo

Tema 1.2 – Introducción al Aprendizaje Profundo

Irina Arévalo

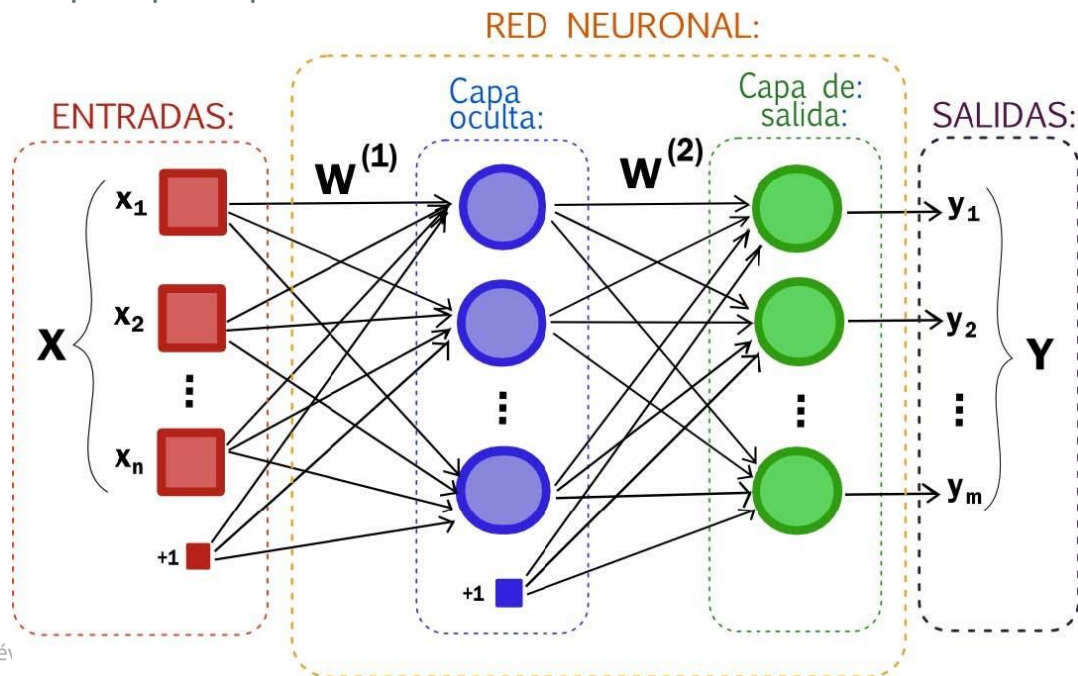


Contenido

1. Machine Learning
2. Perceptrón
3. Redes neuronales
4. Métodos del gradiente
5. Regularización

3. Redes neuronales

- Los perceptrones son algoritmos lineales, las relaciones entre las entradas y las salidas (antes de la función de activación) son combinaciones lineales. Para encontrar **relaciones no lineales** entre las variables, las **redes neuronales (multicapa)** combinan múltiples perceptrones con funciones de activación



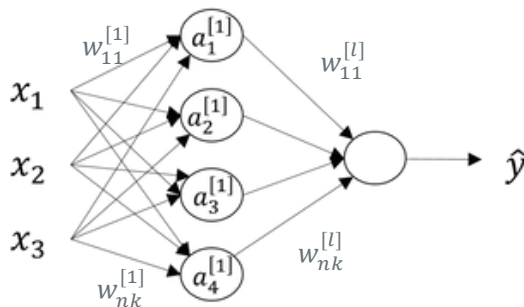
3. Redes neuronales

- Las redes neuronales son una **familia de algoritmos de machine learning paramétricos, no lineales y jerárquicos** que son optimizados con optimización por gradiente.
- La arquitectura de las redes tiene **tres partes**:
 - Una **capa de entrada** que recibe los datos de entrada.
 - **Capas ocultas**, que reciben este nombre porque no son visibles ni desde la entrada ni desde la salida. El número de capas ocultas es variable y depende del objetivo perseguido.
 - **Una capa de salida** que, en función del tipo de escenario en el que nos encontremos (clasificación o regresión) tendrá una o más neuronas.
- La información **fluye desde la capa de entrada hasta la capa de salida**, siendo procesada en las capas ocultas, y la salida de todas las neuronas sirve de entrada para todas las neuronas de la siguiente capa. En general hablaremos de las redes estáticas, pero también existen las redes neuronales dinámicas, que pueden modificar su estructura en función del aprendizaje

3. Predicción

Predicción con redes neuronales

Supongamos que tenemos una red neuronal con l capas:



Dados los inputs $\{x_1, x_2, \dots, x_n\}$ calculamos las entradas de la primera capa oculta $\{a_1^{[1]}, a_2^{[1]}, \dots, a_k^{[1]}\}$ (donde k es el número de neuronas en la primera capa) con una matriz de pesos

$$W^{[1]} = \begin{pmatrix} w_{11}^{[1]} & \dots & w_{1k}^{[1]} \\ \dots & \dots & \dots \\ w_{n1}^{[1]} & \dots & w_{nk}^{[1]} \end{pmatrix}$$

y una función de activación que producirá una relación no lineal.

3. Predicción

Predicción con redes neuronales

Así,

$$a_1^{[1]} = \sigma \left(w_{11}^{[1]} x_1 + \cdots + w_{n1}^{[1]} x_n + b_1^{[1]} \right),$$

donde $b_1^{[1]}$ es un parámetro adicional llamado el sesgo o bias y σ es una función de activación.

Esta fórmula es equivalente para cada neuronal en las capas ocultas

$$a_j^{[1]} = \sigma \left(w_{1j}^{[1]} x_1 + \cdots + w_{nj}^{[1]} x_n + b_j^{[1]} \right)$$

y cada capa

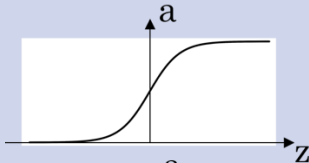
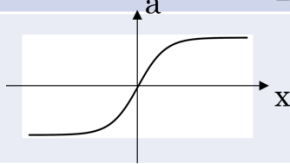
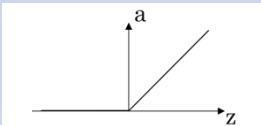

$$a_j^{[l]} = \sigma \left(w_{1j}^{[l]} a_1^{[l-1]} + \cdots + w_{nj}^{[l]} a_n^{[l-1]} + b_j^{[l]} \right).$$

Si l es la última capa, $a^{[l]}$ es la predicción para el input $\{x_1, x_2, \dots, x_n\}$, $a^{[l]} = \hat{y}$.

3. Funciones de activación

Funciones de activación

Algunas de las más comunes son:

Nombre	Fórmula	Gráfica	Descripción
Sigmoide	$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$		También llamada activación logística, muy habitual en clasificación binaria porque devuelve la probabilidad de pertenecer a la clase 1.
Tanh	$f(x) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$		Se suele usar más que la sigmoide en las capas ocultas porque hace que las neuronas estén más cerca de 0
ReLU	$f(x) = \begin{cases} 0, & \text{para } x < 0 \\ x, & \text{para } x \geq 0 \end{cases}$		La función de activación más utilizada
Radial	$h(x) = \exp\left(-\frac{(x-c)^2}{r^2}\right)$		Usa funciones Gaussianas, cada vez menos utilizada

3. Predicción

Capa de salida

La capa de salida es una capa especial porque es la que da el resultado de la predicción. Hay tres tipos de funciones de activación para la capa de salida:

- **Función lineal (o identidad)**: usada para problemas de **regresión**. El output de la red debe ser un **número continuo**, por lo que no se aplica ninguna transformación tras la suma de los productos.
- **Función sigmoide**: usada para problemas de **clasificación binaria**. El output de la red debe ser un **número continuo entre 0 y 1 que represente la probabilidad de que el input pertenezca a la clase positiva**, por lo que la función sigmoide transforma el resultado de la suma de los productos en esos valores entre 0 y 1.
- **Función softmax**: usada para problemas de **multiclasificación**. El output de la red debe ser un **vector con las probabilidades de que el input pertenezca a cada una de las clases posibles**, por lo que la función softmax transforma el resultado de la suma de los productos en ese vector.

3. Predicción

Parámetros e hiperparámetros de una red neuronal

Por lo tanto, los parámetros que tiene la red son:

- El **número de capas** /
- El **número de neuronas** en cada capa oculta
- Los **pesos** W y el **bias** b
- La **función de activación** en las capas ocultas (la función de activación en la salida suele venir dada por el tipo de problema: regresión, clasificación binaria, clasificación múltiple...)
- Como en cualquier modelo de Machine Learning, la **función de coste y las métricas** (de las que hablaremos más adelante sobre la manera de aprender de las redes)

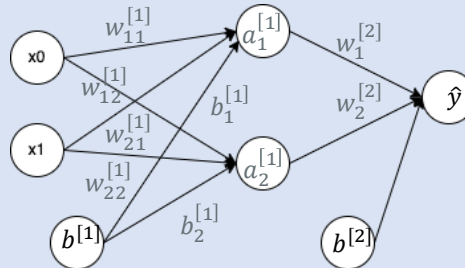
3. Ejemplo

Supongamos que tenemos los siguientes datos:

x_0	x_1	y
1	2	3
2	3	5
3	7	10

donde x_0 , x_1 son las características e y es la variable objetivo (regresión)

Vamos a definir una red neuronal de dos capas con **una capa oculta con dos neuronas**.



3. Ejemplo

Supongamos que los **parámetros iniciales** son:

$$W^{[1]} = \begin{pmatrix} w_{11}^{[1]} & w_{12}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} \end{pmatrix} = \begin{pmatrix} 0.75 & -0.5 \\ 0.5 & -0.25 \end{pmatrix}$$

$$b^{[1]} = \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \end{pmatrix} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}$$

y usaremos la **función de activación sigmoide**.

Entonces:

$$a_1^{[1]} = \sigma(w_{11}^{[1]}x_0 + w_{12}^{[1]}x_1 + b_1^{[1]}) = \sigma(0.75x_0 + 0.5x_1 + 0.2)$$

y

$$a_2^{[1]} = \sigma(w_{21}^{[1]}x_0 + w_{22}^{[1]}x_1 + b_2^{[1]}) = \sigma(-0.5x_0 - 0.25x_1 - 0.2)$$

3. Ejemplo

Para los datos que tenemos:

x0	x1	y	$a_1^{[1]}$	$a_2^{[1]}$
1	2	3	0.88	0.23
2	3	5	0.96	0.12
3	7	10	1	0.03

Y para la segunda capa:

$$W^{[2]} = \begin{pmatrix} w_1^{[2]} \\ w_2^{[2]} \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

$$b^{[2]} = -0.4$$

y la función de activación ReLU.

3. Ejemplo

Entonces

$$\hat{y} = a^{[2]} = \sigma(w_1^{[2]}a_1^{[1]} + w_2^{[2]}a_2^{[1]} + b^{[2]}) = \sigma(2a_1^{[1]} - 1a_2^{[1]} - 0.4)$$

y

x0	x1	y	$a_1^{[1]}$	$a_2^{[1]}$	\hat{y}
1	2	3	0.88	0.31	1.04
2	3	5	0.96	0.18	1.35
3	7	10	1	0.05	1.55

La predicción está muy lejos de los datos reales, por lo que los parámetros W y b no son los adecuados para este problema. A continuación veremos cómo aprende una red neuronal estos parámetros.

3. Ejemplo en Python

- Vamos a implementar este mismo ejemplo en Numpy

```
import numpy as np

X = np.array([[1, 2], [2, 3], [3, 7]])
y = np.array([[3], [5], [10]])
```

- Definimos las funciones sigmoide y ReLU

```
def sigmoid(x):
    return (1 / (1 + np.exp(-x)))

def ReLU(x):
    return x * (x > 0)
```

3. Ejemplo en Python

- Y los pesos (ojo con las dimensiones!)

```
W1 = np.array([[0.75, -0.5], [0.5, -0.25]])  
b1 = 0.2  
  
W2 = np.array([[2], [-1]])  
b2 = -0.4
```

- Entonces la predicción es:

```
def prediccion(X, y, W1, b1, W2, b2):  
  
    Z1 = np.dot(X, W1) + b1  
    A1 = sigmoid(Z1)  
  
    Z2 = np.dot(A1, W2) + b2  
    A2 = ReLU(Z2)  
  
    return(A2)
```

Contenido

1. Machine Learning
2. Perceptrón
3. Redes neuronales
4. Métodos del gradiente
5. Regularización

4. Funciones de coste

- Como comentamos en la introducción sobre algoritmos de Machine Learning, la función de coste de un modelo es **una función que recibe las predicciones y el objetivo y devuelve** un número positivo que representa **la distancia entre ellos**.
- Esta función será **usada para encontrar los parámetros del modelo** (en el caso de redes neuronales, los pesos W y bias b) que **minimizan esta función de coste**, es decir, la diferencia entre las predicciones y el objetivo. Por lo tanto, cuanto más cerca de 0, mejor
- En muchos casos la **función de coste será lineal y por tanto diferenciable**, por lo que podremos usar **métodos clásicos de optimización** para encontrar estos parámetros. Este **no es el caso de las redes neuronales** al introducir partes no lineales en el algoritmo.

4. Funciones de coste

- Existen **multitud de funciones de coste** posibles que serán más adecuadas para unos problemas u otros. Aquí nos centraremos en los más básicos para los problemas de regresión y clasificación binaria.
- Para **regresión**, una métrica y función de coste habitual es el **error cuadrático medio**, **Mean Square Error o MSE** en inglés. Esta métrica encuentra la media entre las diferencias al cuadrado del objetivo y la predicción, por lo que es indiferente al signo de la diferencia. Además la función cuadrada es convexa por lo que podemos encontrar los mínimos globales con métodos clásicos de optimización.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

4. Ejemplo

Volviendo al problema de regresión:

x0	x1	y	\hat{y}	$(y - \hat{y})^2$
1	2	3	1.04	3.84
2	3	5	1.35	13.32
3	7	10	1.55	71.40

Por lo que el MSE es:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 = \frac{1}{3} (3.84 + 13.32 + 71.40) = 29.52$$

4. Funciones de coste

- Para **clasificación binaria**, la función de coste habitual es la **entropía binaria**, en inglés **Binary Cross-Entropy** o Log Loss. Esta función de coste se basa en que las predicciones $\hat{y}^{(i)}$ serán probabilidades y los objetivos en una clasificación binaria serán valores 0/1. En estadística, la entropía mide cómo de diferentes son dos distribuciones, así que tiene sentido usar esta métrica para medir cómo de diferentes son los valores 0 y 1.

$$BCE = \frac{1}{n} \sum_{i=1}^n -(y^{(i)} * \log(\hat{y}^{(i)}) + (1 - y^{(i)}) * \log(1 - \hat{y}^{(i)}))$$

- La intuición detrás de esta fórmula viene de la probabilidad: una manera de tratar la probabilidad de un resultado cuando y es binario es: $(\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}}$. Esta función es difícil de optimizar, por lo que optimizamos su log, que es una función convexa.

4. Funciones de coste

- En general, si $y^{(i)} = 1$ entonces

$$-(y^{(i)} * \log(\hat{y}^{(i)}) + (1 - y^{(i)}) * \log(1 - \hat{y}^{(i)})) = -(1 * \log(\hat{y}^{(i)}) + 0),$$

por lo que minimizar esta función maximiza $\hat{y}^{(i)}$ (hace que $\hat{y}^{(i)}$ esté más cerca de 1)

Si $y^{(i)} = 0$ entonces

$$-(y^{(i)} * \log(\hat{y}^{(i)}) + (1 - y^{(i)}) * \log(1 - \hat{y}^{(i)})) = -(0 + (1 - 0) * \log(1 - \hat{y}^{(i)})),$$

por lo que minimizar esta función maximiza $1 - \hat{y}^{(i)}$ (hace que $\hat{y}^{(i)}$ esté más cerca de 0)

4. Ejemplo

Supongamos que hemos entrenado una red neuronal que tiene como objetivo y resultado los siguientes datos:

y	\hat{y}	$-(y^{(i)} * \log(\hat{y}^{(i)}) + (1 - y^{(i)}) * \log(1 - \hat{y}^{(i)}))$
1	0.98	$-(1 * \log(0.98)) = 0.009$
0	0.05	$-(1-0) * \log(1-0.05) = 0.02$
0	0.03	$-(1-0) * \log(1-0.03) = 0.01$
1	0.87	$-(1 * \log(0.87)) = 0.06$

Por lo que el BCE es:

$$\begin{aligned} BCE &= \frac{1}{n} \sum_{i=1}^n -(y^{(i)} * \log(\hat{y}^{(i)}) + (1 - y^{(i)}) * \log(1 - \hat{y}^{(i)})) \\ &= \frac{1}{4} (0.009 + 0.02 + 0.01 + 0.06) = 0.06 \end{aligned}$$

4. Ejemplos en Python

- En Numpy calculamos el MSE con:

```
mse = np.mean((y - A2)**2)
```

- Y el BCE con:

```
def binary_cross_entropy_loss(A2, y):  
    n = y.shape[0]  
    loss = -(1/n) * np.sum(y*np.log(A2) + (1-y)*np.log(1-A2))  
    return loss
```

4. Optimización de la función de coste

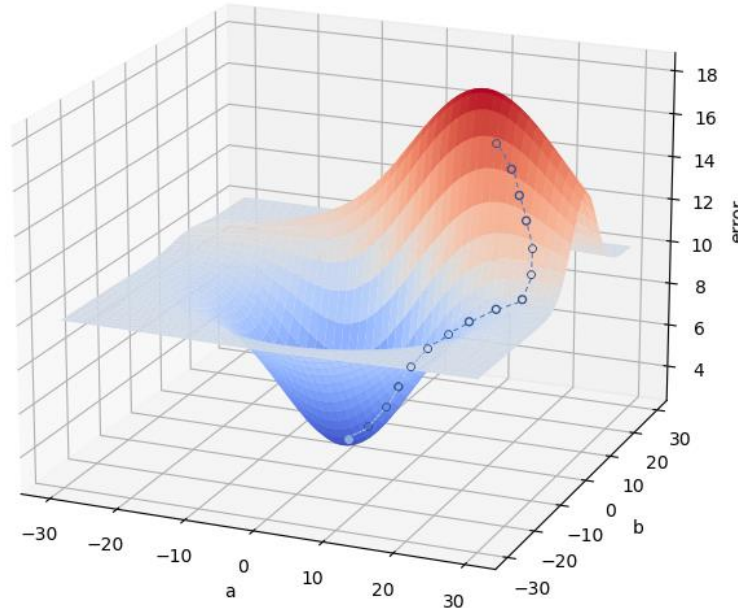
- Como comentábamos, **hay algoritmos para los que optimizar la función de coste es sencillo**. Por ejemplo, supongamos que queremos aproximar nuestros datos mediante una regresión lineal, una función del tipo: $\hat{y}^{(i)} = ax^{(i)} + b$. Para ello necesitaremos encontrar los parámetros a y b que minimizan la función de coste MSE para nuestros datos:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - (ax^{(i)} + b))^2$$

- Al ser el MSE una función cuadrática y $\hat{y}^{(i)}$ una función lineal, es fácil encontrar esos valores óptimos. Sin embargo, **necesitaremos otra táctica para las redes neuronales, que están diseñadas para ser altamente no lineales**.

4. Descenso del gradiente

- El **algoritmo de optimización no lineal** más popular para redes neuronales (y otros modelos de Machine Learning) es el **descenso del gradiente**, un algoritmo que usa las propiedades de las **funciones de coste (convexas)** y de las derivadas.
- Para una función diferenciable, el **gradiente siempre apunta en la dirección de mayor pendiente**, por lo que para encontrar el mínimo sólo debemos descender en esa dirección



4. Descenso del gradiente

- Para obtener los pesos y sesgos óptimos en general **partimos de unos pesos y sesgos aleatorios o 0** (hay múltiples estrategias distintas) y, al igual que para el perceptrón, **actualizamos esos pesos iniciales** con la siguiente estrategia:

$$W = W - \eta \frac{dC}{dW}$$
$$b = b - \eta \frac{dC}{db}$$

donde C es la función de coste y η es la tasa de aprendizaje.



- Esto significa que el descenso del gradiente es un **algoritmo iterativo**, por lo que, para encontrar los pesos y sesgos óptimos **necesitaremos reentrenar múltiples veces**, o **epochs**, la red

4. Backpropagation

- El **cálculo de las derivadas** no es trivial para una red neuronal y se suele utilizar un algoritmo llamado **backpropagation** o retropropagación para **calcularlas desde la capa de salida hasta la capa de entrada usando la regla de la cadena**.

- Sabemos que, suponiendo que hay L capas, la predicción es

$$\hat{y} = \sigma(W^{[L]}a^{[L-1]} + b^{[L]}) = \sigma(z^{[L]})$$

Usando la regla de la cadena tenemos que

$$\frac{dC}{dW^{[L]}} = \frac{dC}{da^{[L]}} \frac{da^{[L]}}{dz^{[L]}} \frac{dz^{[L]}}{dW^{[L]}}$$

y $\frac{dC}{da^{[L]}}$ es la derivada de la función de coste escogida respecto $a^{[L]}$ ($= \hat{y}$), $\frac{da^{[L]}}{dz^{[L]}}$ es la derivada de la función de activación, y $\frac{dz^{[L]}}{dW^{[L]}} = a^{[L-1]}$

4. Backpropagation

- De manera análoga:

$$\frac{dC}{db^{[L]}} = \frac{dC}{da^{[L]}} \frac{da^{[L]}}{dz^{[L]}} \frac{dz^{[L]}}{db^{[L]}}$$

$$\text{y } \frac{dz^{[L]}}{db^{[L]}} = 1.$$

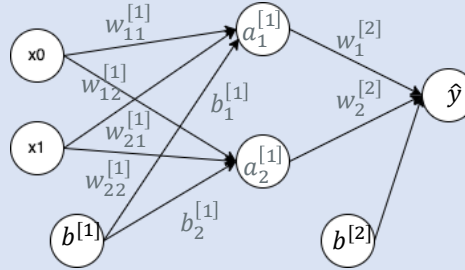
- Continuamos con el mismo procedimiento en cada capa hacia adelante para obtener todas las derivadas parciales $\frac{dC}{dW^{[l]}}$ y $\frac{dC}{db^{[l]}}$ y así actualizar todos los pesos y sesgos.

4. Ejemplo

Volviendo al problema de regresión:

x_0	x_1	y	\hat{y}	$(y - \hat{y})^2$
1	2	3	1.04	3.84
2	3	5	1.35	13.32
3	7	10	1.55	71.40

con esta red



con la función de activación sigmoide en la primera capa y la ReLU en la segunda, y con el MSE como función de coste.

4. Ejemplo

Entonces

$$\frac{dC}{dw_1^{[2]}} = \frac{dC}{da^{[2]}} \frac{da^{[2]}}{dz^{[2]}} \frac{dz^{[2]}}{dw_1^{[2]}} ,$$

donde $\frac{dC}{da^{[2]}}$ es la derivada del MSE con respecto a $a^{[2]}$, que es \hat{y} , por lo que, aplicado a una observación,

$$\frac{dC}{da^{[2]}} = 2(y^{(i)} - \hat{y}^{(i)})$$

Para la segunda parte, $\frac{da^{[2]}}{dz^{[2]}}$, tenemos que recordar que $a^{[2]} = \sigma(z^{[2]})$, por lo que esta derivada es simplemente la derivada de la función de activación, que para la segunda capa es la función ReLU y $\frac{da^{[2]}}{dz^{[2]}} = (z^{[2]} > 0) * 1$

4. Ejemplo

Por último,

$$z^{[2]} = w_1^{[2]} * a_1^{[1]} + w_2^{[2]} * a_2^{[1]} + b^{[2]}$$

por lo que

$$\frac{dz^{[2]}}{dw_1^{[2]}} = a_1^{[1]}$$

y

$$\frac{dC}{dw_1^{[2]}} = \frac{dC}{da^{[2]}} \frac{da^{[2]}}{dz^{[2]}} \frac{dz^{[2]}}{dw_1^{[2]}} = 2(y^{(i)} - \hat{y}^{(i)}) * (z^{[2]} > 0) * 1 * a_1^{[1]}$$

4. Ejemplo

Así, recordando la predicción del primer input, tenemos que

x_0	x_1	y	$a_1^{[1]}$	$a_2^{[1]}$	$z^{[2]}$	\hat{y}	$\frac{dC}{dw_1^{[2]}}$
1	2	3	0.88	0.31	1.04	1.04	3.45

Por lo que el nuevo peso $w_1^{[2]}$ es

$$w_1^{[2]} = w_1^{[2]} - \eta \frac{dC}{dw_1^{[2]}} = 2 - \eta * 3.45,$$

Donde η suele ser un número pequeño, por ejemplo 0.2

Ahora tenemos que repetir el mismo procedimiento para $w_2^{[2]}$ y $b^{[2]}$ (**EJERCICIO**)

4. Ejemplo

Para actualizar los pesos y sesgo de la primera capa, necesitamos encontrar

$$\frac{dC}{dw_{11}^{[1]}} = \frac{dC}{da^{[1]}} \frac{da^{[1]}}{dz^{[1]}} \frac{dz^{[1]}}{dw_{11}^{[1]}},$$

Pero ahora $\frac{dC}{da^{[1]}}$ no es directo. Volvemos a usar la regla de la cadena para llegar a

$$\frac{dC}{da^{[1]}} = \frac{dC}{dz^{[2]}} \frac{dz^{[2]}}{da^{[1]}} = \frac{dC}{da^{[2]}} \frac{da^{[2]}}{dz^{[2]}} \frac{dz^{[2]}}{da^{[1]}},$$

con lo que ya tenemos los dos primeros valores por el cálculo anterior. Para el último, como

$$z^{[2]} = w_1^{[2]} * a_1^{[1]} + w_2^{[2]} * a_2^{[1]} + b^{[2]},$$

entonces

$$\frac{dz^{[2]}}{da^{[1]}} = w_1^{[2]}$$

lo que terminaría el cálculo de $\frac{dC}{da^{[1]}}$.

4. Ejemplo

Volviendo a

$$\frac{dC}{dw_{11}^{[1]}} = \frac{dC}{da^{[1]}} \frac{da^{[1]}}{dz^{[1]}} \frac{dz^{[1]}}{dw_{11}^{[1]}} ,$$

como antes $a^{[1]} = \sigma(z^{[1]})$, por lo que la segunda derivada es la derivada de la función de activación, que para la primera capa es la función sigmoide y **(EJERCICIO)**

$$\frac{da^{[1]}}{dz^{[1]}} = \sigma(z^{[1]})(1 - \sigma(z^{[1]}))$$

Y de nuevo

$$z^{[1]} = w_{11}^{[1]} * x_1 + w_{12}^{[1]} * x_2 + b^{[1]}$$

Y

$$\frac{dz^{[1]}}{dw_{11}^{[1]}} = x_1$$

4. Ejemplo

Así que

$$\begin{aligned}\frac{dC}{dw_{11}^{[1]}} &= \frac{dC}{da^{[1]}} \frac{da^{[1]}}{dz^{[1]}} \frac{dz^{[1]}}{dw_{11}^{[1]}} = \frac{dC}{da^{[2]}} \frac{da^{[2]}}{dz^{[2]}} \frac{dz^{[2]}}{da^{[1]}} \frac{da^{[1]}}{dz^{[1]}} \frac{dz^{[1]}}{dw_{11}^{[1]}} \\ &= \frac{dC}{da^{[2]}} \frac{da^{[2]}}{dz^{[2]}} w_1^{[2]} \sigma(z^{[1]})(1 - \sigma(z^{[1]}))x_1\end{aligned}$$

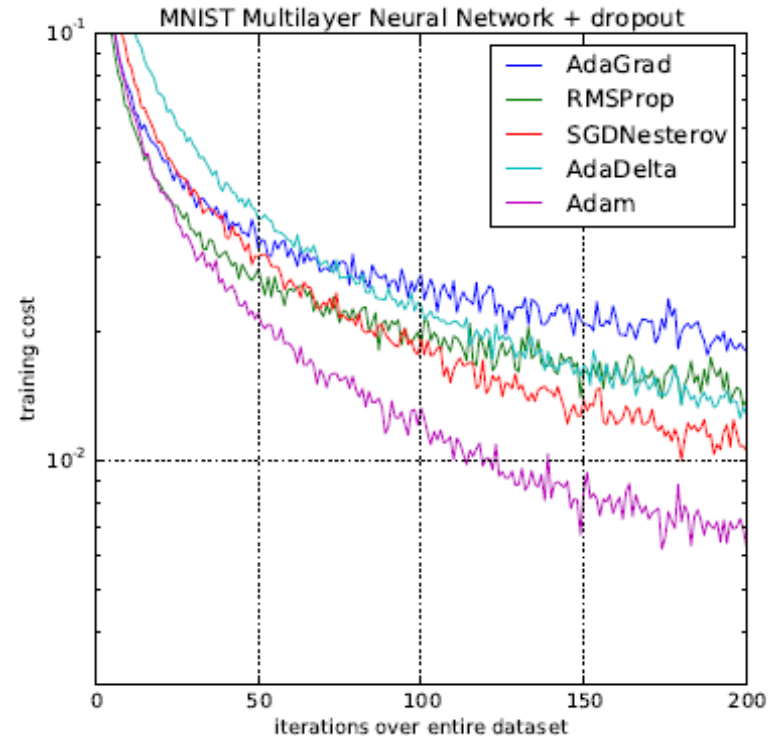
Con lo que actualizamos $w_{11}^{[1]}$ de nuevo con la fórmula

$$w_{11}^{[1]} = w_{11}^{[1]} - \eta \frac{dC}{dw_{11}^{[1]}}$$

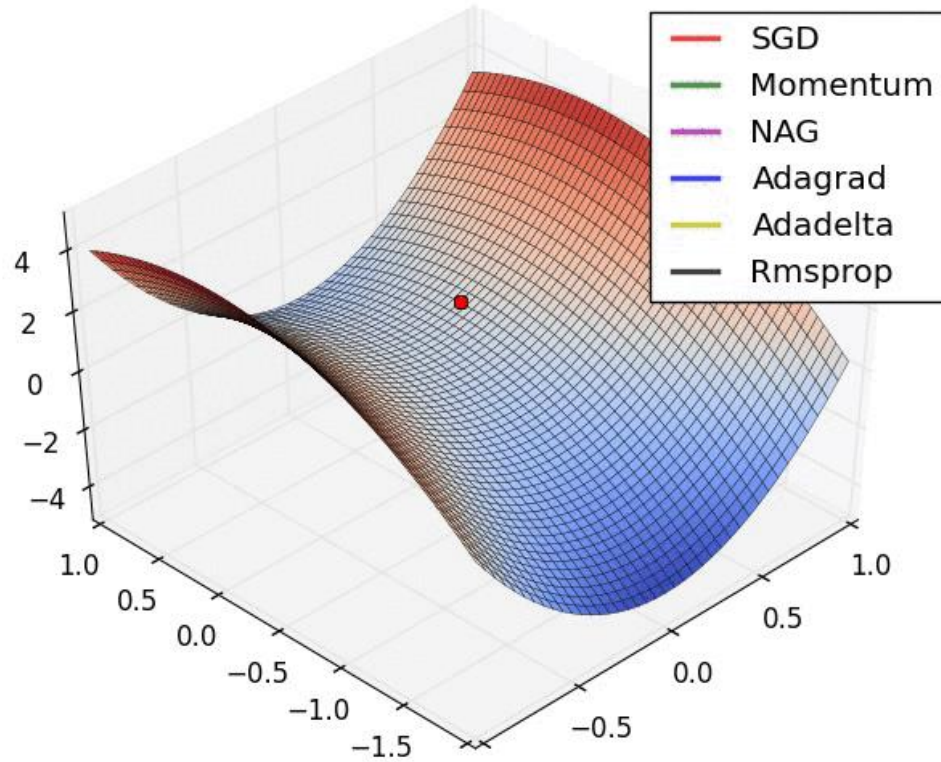
Ahora necesitaríamos también actualizar $w_{12}^{[1]}$, $w_{21}^{[1]}$, $w_{22}^{[1]}$ y $b^{[1]}$ **(EJERCICIO)**

4. Otros métodos de optimización

- Basándose en el descenso del gradiente se han desarrollado distintos métodos de optimización que mejoran al inicial:
 - **Descenso del gradiente estocástico**: actualiza los parámetros de manera más frecuente para alcanzar antes la convergencia
 - **Descenso del gradiente estocástico por mini lotes**: evalúa la función de coste en un subconjunto de los datos de entrenamiento para reducir el coste computacional
 - **Momentum**: añade otro término de aprendizaje para mantener los gradientes acotados
 - **AdaGrad**: la tasa de entrenamiento depende inversamente de los gradientes anteriores para mantenerlos acotados
 - **Adam**: añade un término de decaimiento exponencial



4. Otros métodos de optimización



4. Redes neuronales con Scikit-learn

- La librería Scikit-learn incluye una [implementación de Redes Neuronales](#) muy sencilla
- De nuevo vamos a entrenar un modelo con los datos de [Titanic](#). Como con el perceptrón, vamos a importar MLPClassifier, cargar los datos y dividirlos en train/test

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

df = pd.read_csv('Datos/titanic.csv')
df['Sex'] = df['Sex'].replace({'male': 0, 'female': 1})
X = df.drop(["Name", "Survived"], axis = 1)
y = df["Survived"]
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.8)

model_split = MLPClassifier()
model_split.fit(X_train, y_train)
```

4. Redes neuronales con Scikit-learn

- Evaluamos el rendimiento del modelo con el accuracy en test

```
from sklearn.metrics import accuracy_score  
  
accuracy_score(y_test, model_split.predict(X_test))
```

Output: 0.7808988764044944

- Evaluamos esta red neuronal tan sencilla tiene un mejor accuracy en test que el perceptrón original
- A pesar de que es una implementación muy básica, tenemos varias opciones para mejorar la red neuronal, como el número y tamaño de las capas ocultas (`hidden_layer_sizes`), la función de activación de las capas ocultas (`activation`), el método de optimización (`solver`), las epochs (`max_iter`)...

```
model_improved = MLPClassifier(activation = 'tanh', solver = 'sgd', momentum = 0.99)  
model_improved.fit(X_train, y_train)  
accuracy_score(y_test, model_improved.predict(X_test))
```

Output: 0.8370786516853933

Contenido

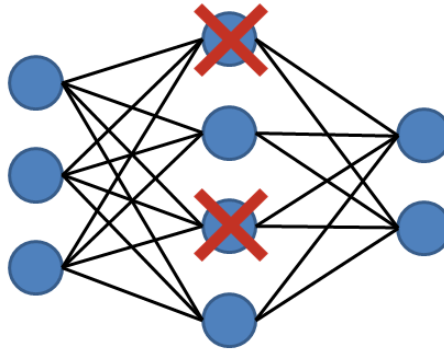
1. Machine Learning
2. Perceptrón
3. Redes neuronales
4. Métodos del gradiente
5. Regularización

5. Regularización

- Todos los modelos de Machine Learning son susceptibles de producir overfitting: acercarse demasiado a los datos de entrenamiento y no ser capaz de generalizar. Sin embargo las redes neuronales son aun más capaces de sobreentrenar.
- Por ello, hay múltiples herramientas para evitarlo como son:
 - Dropout
 - Early stopping
 - Data augmentation

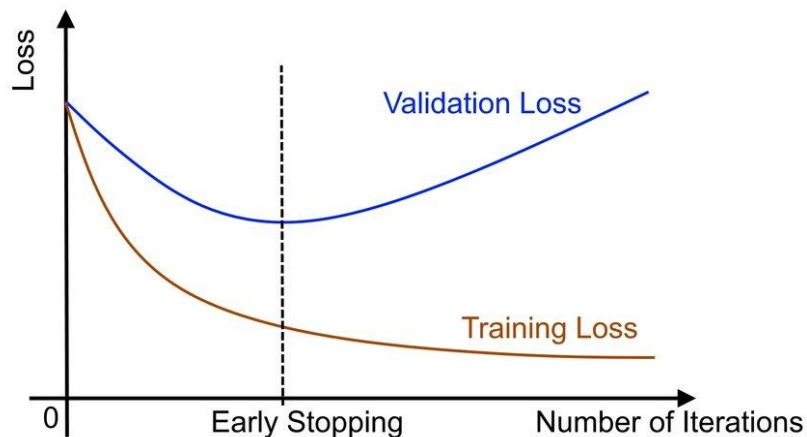
5. Dropout

- Dropout es una herramienta especialmente creada para las redes neuronales
- Durante el entrenamiento, un conjunto aleatorio de neuronas son desactivadas en cada capa
- Gracias a esta retirada de neuronas, el modelo es forzado a ser más robusto y a depender demasiado de un neurona



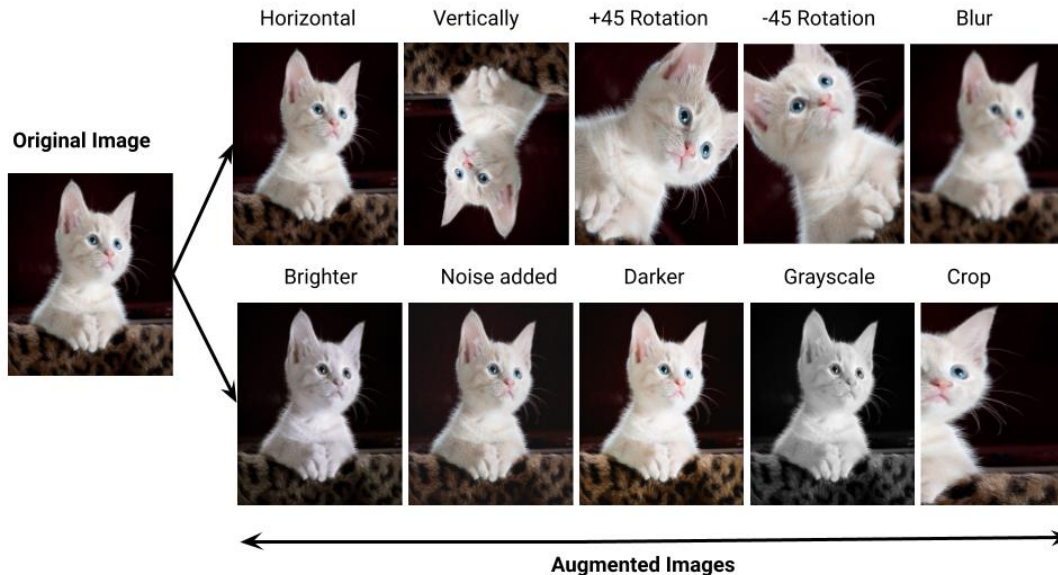
5. Early Stopping

- La parada adelantada es una herramienta que termina el entrenamiento antes de llegar al número de epochs. Cuando el rendimiento del modelo en un conjunto de test deja de mejorar o empieza a empeorar, el entrenamiento se para.



5. Data Augmentation

- El aumento de los datos se usa principalmente en visión por ordenador, donde la cantidad de datos es vital, para crear nuevos datos de entrenamiento al aplicar transformaciones a los datos existentes



Redes Neuronales y Aprendizaje profundo

Tema 1.2 – Introducción al Aprendizaje Profundo

Irina Arévalo

